HW#2 --- last modified Tuesday, 20-Feb-2018 22:04:00 PST.

Solution set.

Due date: Feb 28

Files to be submitted:
Hw2.zip

Purpose:To become more familiar with database record structures, indexes, and B-trees and to experiment with these in a modern DMS

Related Course Outcomes:

The main course outcomes covered by this assignment are:

CLO1 -- Know common database record formats

CLO2 -- Given an index structure based on a B-tree or extensible hashing be able to figure out the effect of performing an insert or a delete

Specification:

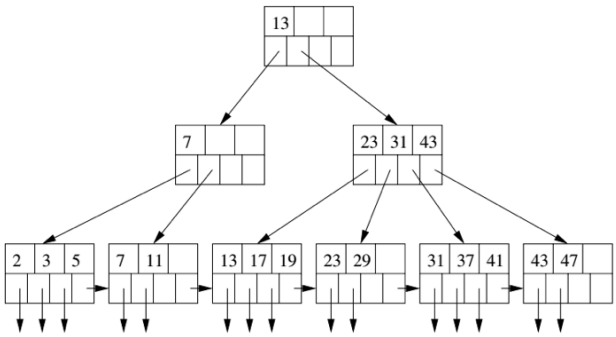
This homework will consist of two parts, a written part and a coding part. Both parts will be submitted in the *Hw2.zip* file. The written part should be in a file *Hw2.pdf*. For the written part of the homework you should write solutions for the following questions. In what you turn in, make sure to write the names and student ids for each group member. For each problem, first copy and paste the question, then write your solution to it beneath it.

1. Suppose that if we swizzle all pointers automatically, we can perform swizzling in a third the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is p, for what values of p is it more efficient to swizzle automatically than on demand?
2. Suppose a record has the following fields in this order: A 4 byte integer, character string of length 11, a SQL date, and a 16 bit Binary field. How many bytes does the record take if:
    a. Fields can start at any byte.
    b. Fields must start at a byte that is a multiple of 4.
3. Suppose we are storing 32 byte records into a data file in our database. Each record has a 4 byte integer key. The data file is sorted by this key. Database blocks are 2048 bytes long. Block pointers are 6 bytes. A record pointer consists of a block pointer followed by two byte number indicating which record offset in the block header is for the record in question. Each block begins with two bytes indicating the number of records in a block, followed by a two byte offset pointing to the first element in the available space list, followed by a list of offsets for each record in the block. Suppose we want to store 50 million records. Calculate showing all work the number of blocks used by the data file and by an index on this data file if:
    a. the index is dense,
    b. the index is sparse.
4. Suppose we have 20000 document collection that has 4,000,000 total words (including repeats). We want

to build an inverted index on this corpus. To keep things simple, we will assume this index is made up dictionary blocks, where a dictionary record consists of a (word,posting-list-pointer) pair; and is made up of posting-list block whose records consist of integer DocIDs. 10 dictionary records can fit in a block, 100 DocIds can fit in a block. Assume the ith most common word occurs with frequency $\frac{120,000}{\sqrt{i}}$ (Zipfian distribution). Given this, calculate:

a. What is the rank of the first word not expected to appear in every document? Words more frequent than this we can treat as stop words.
b. What's the smallest n such that $\sum_{i=1}^{n} \frac{120,000}{\sqrt{i}} \geq 4,000,000$? This can be used to estimate the number of distinct words, we'd expect to see in our corpus.
c. Using the above, how many blocks would you expect the inverted index dictionary to take?
d. How many blocks would you expect the posting list for the 400th most common word to take?

5. For the following B-tree, show:
   a. step-by-step the blocks accessed in looking up 12 (a key not in the tree)
   b. step-by-step inserting a new entry for a record with key 35



For the coding part of the assignment, I'd like you to programmatically implement grid files in Sqlite using a java program named *GridFileManager.java*. To keep things simple, we will assume our grid files are used to store triple (x,y, label) data. To test your code, the grader will compile it from the command line with a line like:

```
javac GridFileManager.java
```

Your code should include two other classes given by the files *GridPoint.java* and *GridRecord.java*. Here is all the code you need for these classes:

```
//GridPoint.java
public class GridPoint
{
    public float x;
    public float y;
    public GridPoint(float x, float y)
    {
        this.x = x;
        this.y = y;
    }
    public String toString()
    {
        return "(" + x + "," + y + ")";
    }
}
//GridRecord.java
```

```
public class GridRecord
{
    public String label;
    public GridPoint point;
    public GridRecord(String label, GridPoint point)
    {
        this.label = label;
        this.point = point;
    }
    public GridRecord(String label, float x, float y)
    {
        this.label = label;
        this.point = new GridPoint(x, y);
    }
    public String toString()
    {
        return label + point.toString();
    }
}
```

Your program *GridFileManager* will be run from the command line with a line like:

```
java GridFileManager sqlite_database_name name_of_instruction_file
```

For example,

```
java GridFileManager test.sqlite instructions.txt
```

The above command should create a connection to the Sqlite database *test.sqlite* and then execute each of the instructions in *instructions.txt* on that database. An instruction file consists of a sequence of lines, each line being an instruction. These can be one of three types:

1. **A create grid file line** which creates a new grid file. It has the format:

   ```
   c file_name low_x high_x num_lines_x low_y high_y num_lines_y num_buckets
   ```

   It creates a new 2D grid file named *file_name*. The grid lines in the *x*-axis (resp *y*-axis) begin with the line x=low_x (resp. y=low_y) and end with the line x=high_x (resp. y=high_y) and there are exactly num_lines_x (resp num_lines_y) many equally spaced lines. You can assume that num_lines_x and num_lines_y are always at least 2. The following is an example of a create instruction:

   ```
   c SuperGrid -5 5 3 0 10 3 100
   ```

2. **An insert grid file line** which inserts a record into a grid file. It has the format:

   ```
   i file_name label x_value y_value
   ```

   This instruction should insert into the grid file *file_name* (only if it exists) the record given by *label x_value y_value*. The following is an example of an insert instruction:

   ```
   i SuperGrid IAmHere -3 6
   ```

   This should insert into the grid file *SuperGrid* a record with label *IAMHere* at location (-3,6).
3. **A lookup grid file line** which looks up the records in the grid file contained within a rectangle given by two points. It has the format:

   ```
   l file_name p1_x p1_y p2_x p2_y limit_offset limit_count
   ```

This should output a sequence of lines listing *limit_count* many records in the grid file *file_name* beginning *limit_offset*th record found that belongs to the rectangle given by the pair of points (p1_x, p1_y) and (p2_x p2_y). The following is an example of a lookup instruction:

```
l SuperGrid -5 1 10 10 5 10
```

This should output at most 10 records beginning with the 5th such record found that lies in the rectangle given by the points (-5, 1) and (10,10).

To implement the above instructions, the *GridFileManager.java* file should contain a class *GridFileManager* which has the following public methods which roughly correspond to each of these operations:

1. **GridFileManager(String databaseName)** -- the constructor takes the name of the Sqlite database, the GridFileManager is supposed to manage the grid files for. It then creates the following four tables in this database if they don't exist:
    a. GRID_FILE(ID INTEGER PRIMARY KEY, NAME VARCHAR(64), NUM_BUCKETS)
    b. GRIDX(GRID_FILE_ID INTEGER PRIMARY KEY, LOW_VALUE INTEGER, HIGH_VALUE INTEGER, NUM_LINES INTEGER)
    c. GRIDY(GRID_FILE_ID INTEGER PRIMARY KEY, LOW_VALUE INTEGER, HIGH_VALUE INTEGER, NUM_LINES INTEGER)
    d. GRID_FILE_ROW(GRID_FILE_ID INTEGER, BUCKET_ID INTEGER, X REAL, Y REAL, LABEL CHAR(16), PRIMARY KEY(GRID_FILE_ID, BUCKET_ID))
2. **boolean createGridFile(String fileName, int lowX, int highX, int numLinesX, int lowY, int highY, int numLinesY, int numBuckets)**. This method should insert a row into the table GRID_FILE with values fileName for the name and numBuckets for NUM_BUCKETS. It should then insert one row into GRIDX and one row into GRIDY using the rest of its arguments. The return value of this method should indicate if it succeeded to do the inserts or not.
3. **boolean add(String fileName, GridRecord record)**. This function should insert the passed *GridRecord* into the grid file given by *fileName*. To do this it should determine the ID in GRID_FILE for *fileName* and the NUM_BUCKETS of this grid file. Then using the GridPoint in GridRecord, and the appropriate rows in GRIDX and GRIDY, determine which rectangle in the grid file the GridPoint belongs to. Then after applying a hash function with range between 0 and NUM_BUCKETS -1, determine which bucket to add the GridRecord. Given this, it should add the appropriate row to GRID_FILE_ROW.
4. **GridRecord[] lookup(String fileName, GridPoint pt1, GridPoint pt2, int limit_offset, int limit_count)** This function should return an array of up to *limit_count* (if they exists) GridRecord's from the grid file *fileName* beginning with the *limit_offset*th such record found that lies in the rectangle given by the points *pt1* and *pt2*. To do this your code should using the tables GRID_FILE, GRIDX, and GRIDY determine which grid rectangles intersect the retangle to look up, then for each record in these rectangles, check if it belongs to the lookup rectangle and if so add it to the return array.

This completes the description of the program you need to write.

Point Breakdown

Written problems 1 (0 not close, 1/2 partially correct, 1 fully correct).	1pt
Written problems 2,3, 5 (1pt, 1/2pt/per sub-part).	3pts
Written problems 4 (1/2pt per sub-part).	2pts
Program compiles and when run processes instruction file on sqlite database given by the command line arguments	1pt

as described	
GridFileManager constructor works as described	0.5pts
createGridFile method works as described	0.5pts
add method works as described	1pt
lookup method works as described	1pt
Total	10pts